JAVASCRIPT

Michael Gfeller (mgfeller@hsr.ch)

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

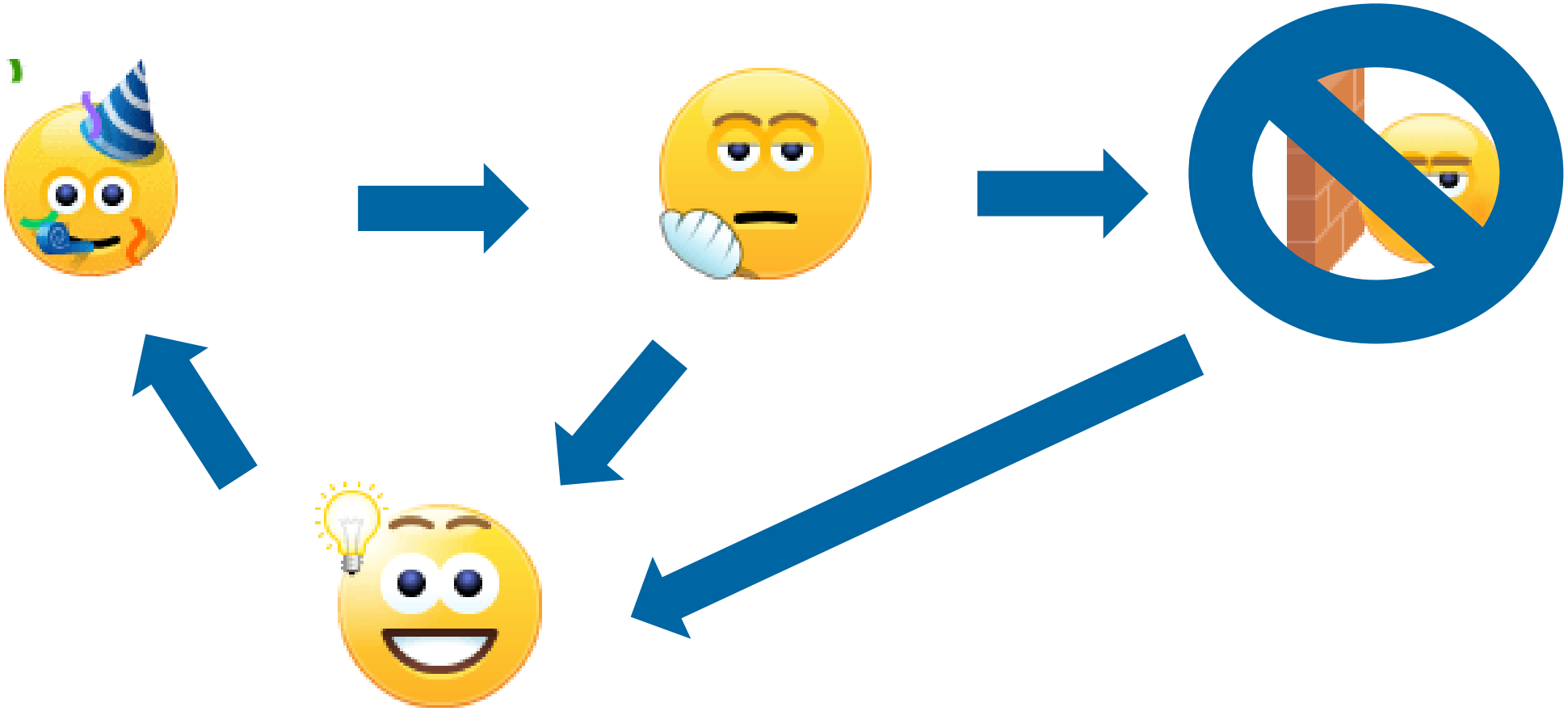
FHO Fachhochschule Ostschweiz

Intro

- **JavaScript ist mächtig**
- **JavaScript ist super**
- **JavaScript wird immer besser**
- **JavaScript ist gefährlich**

Bild-Quelle: http://engineering.wix.com/wp-content/uploads/2015/04/mIuuwgx-1024x576.jpg

Das Leben mit JavaScript (und allgemein mit Web-Technologien)



Inhaltsverzeichnis

https://github.com/gfeller/Vorlesung_JS

Lernziele

Die Teilnehmer…

- **… können JavaScript programmieren**
- **… kennen die Spezialitäten / Pitfalls von JavaScript**
- **… sind in der Lage eine JS Code Guideline nachzuvollziehen und sinnvolle Punkte zu extrahieren**
- **… kennen den Unterschied zwischen ECMAScript und JavaScript**
- **… können Array-Funktionen anwenden und kennen die Unterschiede zwischen den verschiedenen Iterationsvarianten**
- **… können die neuen Features von ECMAScript 6 anwenden**
- **… können ein neue JavaScript Feature auch in alten Browsern zu Verfügung stellen**
- **… können (automatische) Typenumwandlungen nachvollziehen und erklären**

EINSTIEG

ECMAScript vs. JavaScript

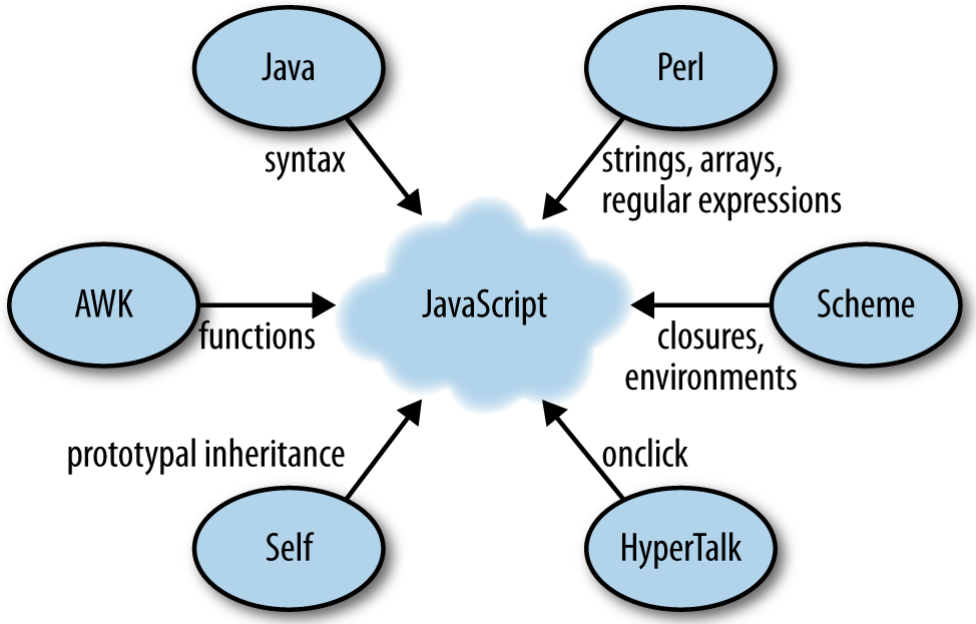
- **ECMAScript ist der offizielle Name für JavaScript**
- **Oracle/Sun haben die Rechte an dem Namen «JavaScript»**

Generell:

- **JavaScript wird für die Programmiersprache verwendet**
- **ECMAScript ist der Name für die Sprachdefinition**
  - z.B. Die aktuelle Version von JavaScript ist ECMAScript 2018 (9) und ECMAScript 2019 wird entwickelt
  - Siehe https://kangax.github.io/compat-table/es2016plus/ für Browsersupport

JavaScript Erfolgsgeschichte

1995	JavaScript in 10 Tagen entwickelt
1997	Dynamic HTML: HTML kann geändert werden
1999	XMLHttpRequest: Nachträgliches laden von Daten vom Server
2001	JSON
2005	Google Maps erscheint: Perfektionierte Dynamic HTML und XMLHttpRequest. => AJAX
2006	JQuery
2007	WebKit
2008	V8 – Engine (Chrome)
2009	NodeJs
2009	PhoneGap / ChromeOS



ECMAScript Versionen

Version	Publiziert	Unterschiede zur Vorgängerversion
1, 2	1997, 1998	Erste Version, Änderungen zwecks Kompatibilität zum internationalen Standard ISO/IEC 16262
3	1999	Neu sind reguläre Ausdrücke, bessere Verarbeitung von Zeichenketten, Kontrollfluss, **Fehlerbehandlung mit try/catch**, bessere Fehlerbehandlung, bessere Formatierung bei der Ausgabe von Zahlen usw.
4	abgebrochen	Wegen Uneinigkeit in Bezug auf die Zukunft der Sprache eingestellt. Einige Ideen werden in ES6 wieder aufleben.
5	Dezember 2009	Im „**strict mode**“ wird eine erweiterte Fehlerprüfung eingeschaltet. Unklare Sprachkonstrukte von ECMAScript 3 werden entschärft und neue Features wie getter- und setter-Methoden, Unterstützung von **JSON** usw. hinzugefügt.
5.1	Juni 2011	Entspricht dem internationalen Standard **ISO/IEC 16262:2011, Version 3**
6 / 2015	Juni 2015	**Neue Syntax für komplexe Applikationen wie Klassen und Module**, die aber mit ähnlicher Terminologie wie in ECMAScript 5 (strict mode) definiert werden können. Neue Sprachbestandteile wie for/of-Schleifen, teilweise an Python angelehnte Syntax usw. Der Codename lautet “Harmony” und wurde bis kurz vor Verabschiedung als „ECMAscript 6“ bezeichnet
2016 / 7	Juni 2016	** Operator und Array.prototype.includes
2017 / 8	Juni 2017	New features proposed include concurrency and atomics, zero-copy binary data transfer, more number and math enhancements, syntactic integration with promises (**await/async**), observable streams, SIMD types, **better** metaprogramming with **classes**, class and instance properties, operator overloading, value types (first-class primitive-like objects), records and tuples, and traits.
2018 / 9	Juni 2018	features for **asynchronous iteration and generators**, new regular expression features and rest/spread parameters.

Quelle: https://en.wikipedia.org/wiki/ECMAScript#Versions

ECMAScript 6 / 2015

- **Grösster Schritt von JavaScript**
- **Grundlage der Vorlesung**
- **Unterstützt von allen modernen Browsern**
- **Support von neuen ECMAScript-Versionen in älteren Browsern mit Hilfe von «JavaScript Compiler»**
  - https://babeljs.io/
  - https://github.com/google/traceur-compiler
- **Zusammenfassung der ECMAScript 6 Features mit Beispielen**
  - http://es6-features.org/

JAVASCRIPT AUSFÜHREN

JavaScript im Browser ausführen

- **JavaScript wird üblicherweise ins HTML eingebunden.**

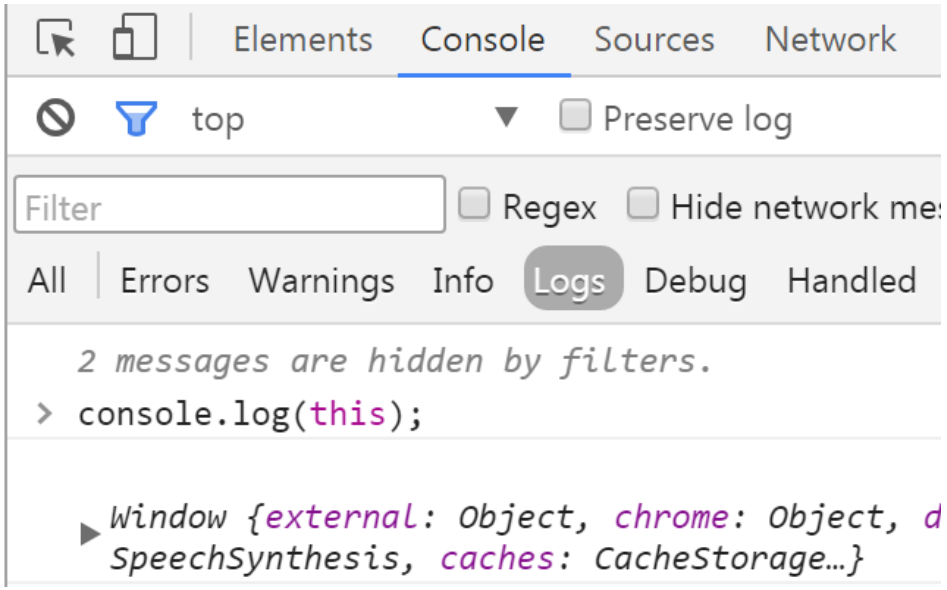
```
<!-- Variante 1a: PRE HTML5 -->
<script type="text/javascript" language="javascript">
    alert("TEST");
</script>


<!-- Variante 1b: HTML5 -->
<script>
    alert("TEST");
</script>


<!-- Variante 2: Externe Datei -->
<script src="01_HelloWorld.js"></script>


<!-- Variante 3: HTML Attribute -->
<button onclick="alert('hi hsr')">
```

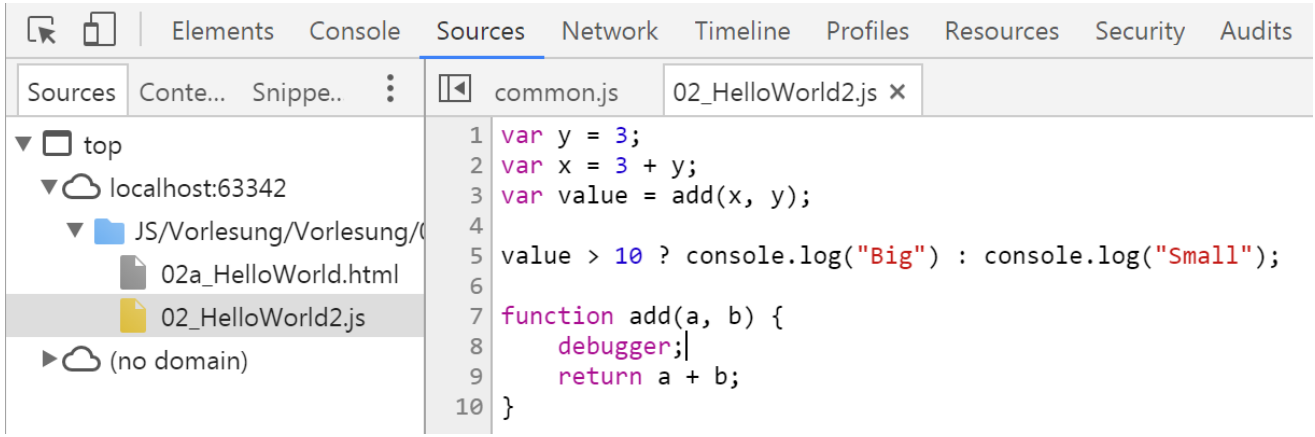
Script Tag: http://www.w3.org/TR/html5/scripting-1.html#script

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <!-- 1 -->
</head>
<body>
<!-- 2 -->
<h1>content</h1>
<!-- 3 -->
</body>
</html>
```

JavaScript Dev. Tools

- **In jedem grösseren Browser vorhanden**
- **Sehr mächtig**
- ***Tipp: Unbedingt nutzen!***





JavaScript in Node.js ausführen

- **Kann JavaScript ausführen und debuggen**
  - Nutzt die V8 Engine von Chrome
  - Wird von VS Code genutzt, um JavaScript auszuführen
- **Installation https://nodejs.org/en/**
  - Kann über die Command-Line angesteuert werden
- **Folgende Parameter aktivieren unstable Features z.B.:**
  - --harmony
  - --harmony_trailing_commas
  - Weitere Informationen: https://nodejs.org/en/docs/es6/
- **Node.js kann als Web Server fungieren => später**

Erstes Beispiel ausführen

```
let y = 3;
let x = 3 + y;
const value = add(x, y);

(value > 10) ? console.log("Big") : console.log("Small");

function add(a, b) {
    debugger;
    return a + b;
}
```

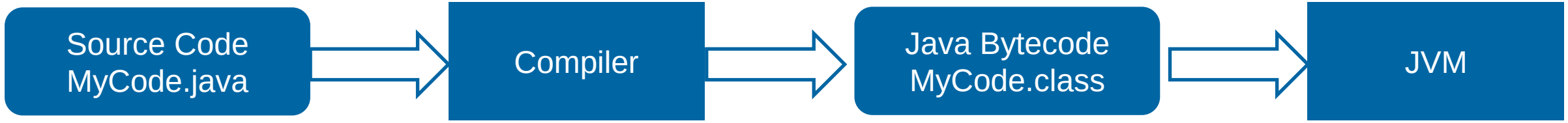
JAVASCRIPT

Die Natur von JavaScript

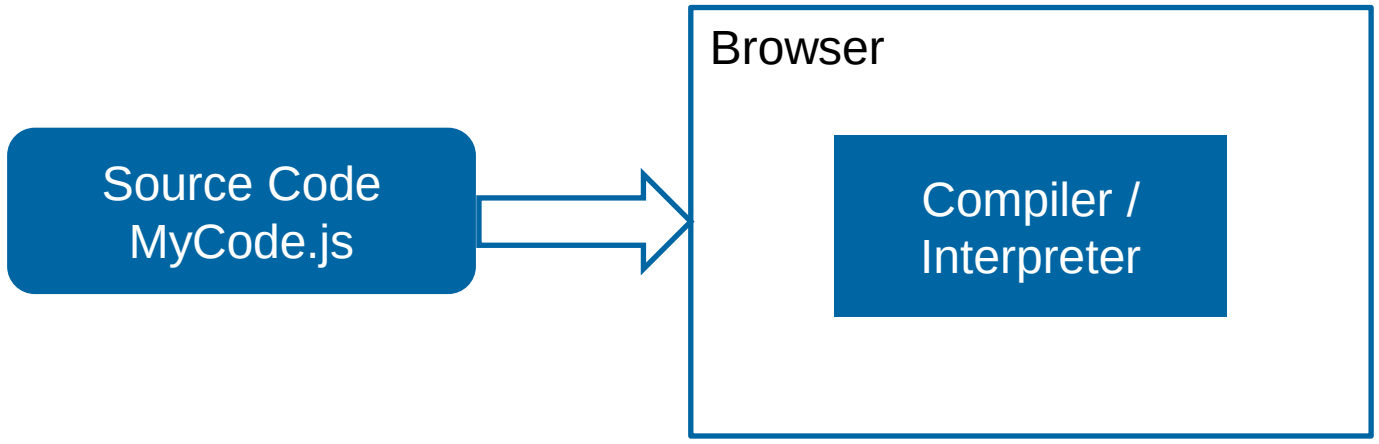
- **It's dynamic**
  - Objekte können verändert werden z.B. Methoden überschrieben werden
- **It's dynamically typed**
  - Variablen können den Type ändern – je nach Inhalt
- **It's functional and object-oriented**
- **It fails silently**
  - Bei Fehler wird oft keine Exception geworfen sondern läuft weiter. z.B. 0/0 = NaN (Not a Number)
- **It's deployed as source code**
  - JavaScript wird erst beim Ziel (z.B. Browser) interpretiert bzw. kompiliert
- **It's part of the web platform**
  - Auch ohne Browser lauffähig
- **In einigen Belangen anders als Java!**

It’s deployed as source code

Java Code



JavaScript Code



JAVASCRIPT-TYPEN

It’s dynamically typed

JavaScript is a dynamically typed language.

- **Variablen benötigen keine Typendeklarierung**
- **Die gleiche Variable kann über die Zeit unterschiedliche Typen beinhalten**
  - Für Nachvollziehbarkeit sollte einer Variable immer nur Werte vom gleichem Typen zugewiesen werden!
- ***typeof()* kann genutzt werden um den Type der Variable abzufragen**

```
let foo;
foo = "Michael";
console.log(foo + " is a " + typeof(foo));
foo = 42;
console.log(foo + " is a " + typeof(foo));
foo = true;
console.log(foo + " is a " + typeof(foo));
```

```
Output:


Michael is a string

42 is a number

true is a boolean
```

Typen-System

Wie Java unterscheidet auch JavaScript zwischen Primitives und Objekten

- **Primitive Typen**
  - string; number; boolean; null; undefined; symbol (ECMAScript 6)
  - Compared by value
  - Always immutable

- **Objekte**
  - Alles andere: Plain Objekts, Arrays, Regular Expressions, Functions
  - Compared by reference
  - Mutable by default

Typeof

typeof(type)	Result
Undefined	'undefined'
Null	'object'
Boolean	'boolean'
Number	'number'
String value	'string'
Function	'function'
Symbol (ECMAScript 6)	‘symbol'
All other	'object'

PRIMITIVES

Booleans

- **`true` und `false`**
- **Jeder Wert kann in ein boolean gewandelt werden**
  - !!(null) => false
  - Boolean(null) => false
- **Logische Operatoren**
  - And: &&
  - Or: ||
- **Prefix Operatoren**
  - Not: !
- **Vergleichsoperatoren**
  - ===, !==, ==, !=
    - Achtung, Unterschiede zu Java! Mehr dazu später in der Vorlesung.
  - >, >=, <, <=

Falsy / Truthy

`false`-Werte:

- false
- 0 (zero)
- "" (empty string)
- null
- undefined
- NaN

`true`-Werte:

- Alles andere
  - "0" (string)
  - "false" (string)
  - []
  - {}
  - …

```
console.log(Boolean(undefined));   //false
console.log(Boolean(0));           //false
console.log(Boolean(3));           //true
console.log(Boolean({}));          //true
console.log(Boolean([]));          //true
```

Number

- **Nach Definition sind alle Zahlen «floats» (Gleitkommazahlen)…**
- **Die Engines versuchen die floats auf integers abzubilden – falls möglich**

```
console.log(0.3333333333333333 * 3 == 1);     //true
let x = 1/3;
console.log(x + x + x);                       //1
console.log(0.3 + 0.7);                       //1
console.log(Number.isInteger(x + x + x));     //true
console.log(Number.isInteger(0.3));           //false
console.log(0.1 + 0.2);                       //0.3000000000000004
```

  - Funktioniert nicht immer. Grund: Ist ausserhalb vom gültigen Bereich

```
console.log(99999999999999999); //100000000000000000
Number.isSafeInteger(99999999999999999); //false
```

- Definition: http://de.wikipedia.org/wiki/IEEE_754
- Beschreibung: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Number

Number

- **NaN («Not a Number»)**
  - Ist ein Error-Wert
  - 0/0 => NaN
  - Hat auch den Type «number»
  - NaN == NaN ist immer false
    - isNaN( ) zum Überprüfen
- **Infinity**
  - Unendlich
  - Kann auch negativ sein

```
let div0 = 0 / 0;
console.log( div0);                //NaN
console.log( typeof(div0));        //number
console.log( parseInt("abc"));     //NaN
console.log( div0 == NaN);         //false
console.log( isNaN(div0));         //true


console.log( 3 / 0);                 //Infinity
console.log( Math.pow(2,10000));     //Infinity
console.log( -Math.pow(2,10000));    //-Infinity
```

Number

- **Jeder Wert kann in eine Zahl verwandelt werden**
  - *+(true) == 1*
  - *Number(true) == 1*
  - *Number(null) == 0*
  - *Number(«abc») => NaN*
  - Ausnahme: *Symbol*
- **parseInt(«string»)**
  parseFloat(«string»)
  - Parst bis zum ersten Fehler

```
console.log( +(true));                  //1
console.log( +(false));                 //0
console.log( +("1ab"));                 //NaN
console.log( +("123"));                 //123
console.log( +([]));                    //0
console.log( parseInt("1.2ab"));        //1
console.log( parseFloat("1.2ab"));      //1.2
console.log( parseInt("abc"));          //NaN
```

String

- **Mit "Text" oder 'Text'**
- **Escape mit «\»**
- **Typische Properties / Methoden vorhanden**
  - length
  - slice()
  - trim()
  - includes()
  - indexOf()

```
'abc'
"abc"

'Did she say "Hello"?'
"Did she say \"Hello\"?"

'That\'s nice!'
"That's nice!"

'Line 1\nLine 2'  // newline
'Backslash : \\'
```

String – Template Strings (ECMAScript 6)

- **ECMAScript 6 brachte ‘Template strings’**
  - Ermöglicht Strings mit Placeholders (und mehr)
  - https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/template_strings
- **String-Typ wird mit `umschlossen`**
  - ` = *backquotes oder backticks*
- **Inhalt innerhalb von ${ … } wird JavaScript interpretiert**
  - Linebreaks und Leerzeichen werden beibehalten

```
const name = "Michael",
      hobby = "Hike",
      a = 4, b = 5;

console.log(`Mein Name ist: ${name}
Hobby: ${hobby}`);

console.log(`${a} + ${b} = ${a + b}`);
```

```
// Früher:
console.log("Mein Name ist: "+name +
            "\nHobby: " + hobby);
```

```
Output:
Mein Name ist: Michael
Hobby: Hike
4 + 5 = 9
```

Aufgabe + - * /

- **Was geben folgende Ausdrücke aus?**

```
"4" / "2"

"4" - "2"

"4" * "2"

"4" + "2"

10 * 3 + "px"

"px" + 1 - 2

1 / 0

"3px" + 3 * 2 + "3px"

"foo"+ +"abc"

"2"- -1
```

- **Zeit: 5 Minuten**

+ - * / Regeln

- **Punkt vor Strich**
- **Von Links nach Rechts aufgelöst**
- **Spezialfälle:**
  - String + Value = String
  - Value + String = String
- **Ansonsten:**
  - Value [Nummerische Operator] Value = Number
  - d.h. + - * / %

The Abstract Equality Comparison Algorithm

- **Nicht immer bringt ein ==-Vergleich die erwarteten Resultate:**

```
console.log([] == false);           //true

console.log("" == false);           //true

console.log(null == false);         //false

console.log(0 == "0");              //true

console.log(null == undefined);     //true

console.log([1,2] == "1,2");        //true

console.log(NaN == NaN);            //false

console.log([] == ![]);             //true
```

The Abstract Equality Comparison Algorithm

- **The Abstract Equality Comparison Algorithm (==)**
  - Beschreibt, wie ein Gleichheitsvergleich von zwei Werten abläuft
  - Besonders aufgrund der dynamischen Typen in JavaScript
  - Resultiert in *true* oder *false*
  - Definition: http://ecma-international.org/ecma-262/#sec-abstract-equality-comparison
  - Beschreibung: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

- **The Abstract Relational Comparison Algorithm (<)**
  - Beschreibt, wie ein Grössenvergleich von zwei Werten abläuft
  - Resultiert in *true* oder *false* (oder *undefined* bei NaN-Werten)
  - Definition: http://ecma-international.org/ecma-262/#sec-abstract-relational-comparison

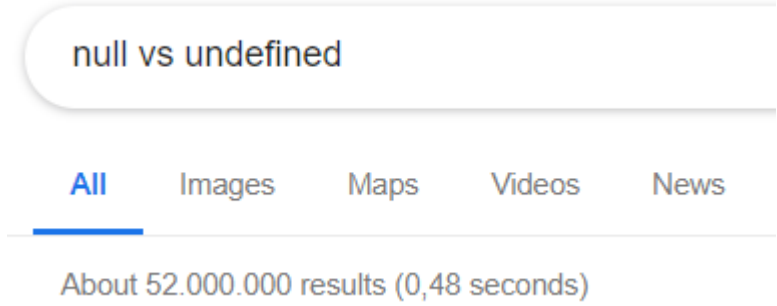
The Abstract Equality Comparison Algorithm

- **===**
  - Verhindert die Typenumwandlung von Primitives
  - Für Objekte nicht notwendig
  - Im Zweifelsfall immer verwenden

```
console.log(false === false);        //true
console.log(4 === 4);                //true
console.log(false === false);        //true

console.log([] === false);           //false
console.log("" === false);           //false
console.log(null === false);         //false
console.log(0 === "0");              //false
console.log(null === undefined);     //false
console.log([1,2] === "1,2");        //false
console.log(NaN === NaN);            //false
console.log([] === ![]);             //false
```

null != undefined

- **JavaScript hat 2 Varianten um «nonvalues» darzustellen**
- **undefined**
  - Variable ist nicht definiert. Z.B. `let a` wurde vergessen
  - Variable ist nicht initialisiert. Z.B. `a = 1234` wurde vergessen
- **null**
  - Ist ein Wert von einer Variable
  - Wird als «nonvalue» verwendet, falls ein Objekt erwartet wird
  - Ähnlich wie *null* in Java
- **Beispiel**
  - Funktionen welche «undefined» zurückgeben, haben keinen Rückgabewert.
  - Funktionen welche «null» zurückgeben, hätten einen Rückgabewert aber nicht in diesem Falle.



- **Achtung**: `null == undefined` **resultiert in** `true`**!**

null != undefined Praxis-Beispiel

- Programm möchte Partielles Updaten ermöglichen. Z.B. nur den Namen anpassen.
  - null => Wert wird auf null gesetzt.
  - undefined => Bestehender Wert wird beibehalten.

```
const userList = require('./user');

console.log( userList.get(0) );
console.log( userList.update(0, {birthday: "19.05.1986"}) );
console.log( userList.update(0, {name : undefined, birthday: "19.05.1986"}) );
console.log( userList.update(0, {name : null, birthday: "19.05.1986"}) );
```

```
{ name: 'Michael', birthday: '19.05.1985' }
{ name: 'Michael', birthday: '19.05.1986' }
{ name: 'Michael', birthday: '19.05.1986' }
{ name: null, birthday: '19.05.1986' }
```

null != undefined

Wie würden Sie überprüfen ob eine Variable undefined ist?
Z.B. myVariable

Wie würden Sie überprüfen ob ein Wert auf einem Objekt undefined ist?

z.B. myVariable.a

```
A:
typeof myVariable == 'undefined';

B:
myVariable.a == undefined;

Combined:
typeof (myVariable) != 'undefined' && myVariable.a == undefined;

Performance:  http://jsperf.com/undefined-null-typeof
```

ARRAY

Array

- **«Klassisches» Array**
- **Keine fixe Länge**
- **Index beginnt bei 0**

```
const arr = [ 'a', 'b', 'c' ];
arr[0] = 'x';
arr.push("d");
console.log(arr); // [ 'a', 'b', 'c', 'd' ]
console.log(arr.length); // 4
```

- **Methoden:**
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Instance_methods

Array Methoden & Iterieren

- **Array bietet Iterator-Methoden an**

```
const arr = [ 'a', 'b', 'c' ];
arr.forEach((elem, index) => console.log(index +":"+elem));

const numberArr = [1,2,3,4].map(x => x * x);
console.log(numberArr);
console.log(numberArr.filter(elem => elem > 5));
console.log(numberArr.every(elem => elem > 5));
```

Array Iterieren mit for of/in

- **«Klassische» for-Schleife**

```
for(let i=0; i<arr.length; ++i) {
    console.log("for",arr[i]);
}
```

- **For-In iteriert über die Property Namen**

```
for(let x in arr) {
    console.log("for in", x + ":" + arr[x]);
}
```

- **For-Of iteriert über die Werte**

```
for(let y of arr) {
    console.log("for of", y);
}
```

```
const dummy = {name: "Hallo", date : Date.now()};

for(let x in dummy)
{
    console.log("dummy has property", x);
}
```

OBJECT

Simple Object

- **Ein Object ist eine Sammlung von Properties.**
- **Die Properties werden mit einem Set (HashSet) verwaltet**
  - Key = String
  - Value = Value
    - boolean / function / string / …
- **Objekt können als «object literals» erstellt werden**
  - Können im Nachhinein mit Properties ergänzt werden

```
const person = {
    name : "Michael",
    hallo : function(){
        return "Hallo "+this.name;
    }
};

person.name = "Bob";
```

```
const myObj = {};
myObj.name = "Michael";
myObj.hallo = function() {
    return `Hallo ${this.name}`
};


console.log(myObj.hallo());
```

Simple Object: It's dynamic

- **Properties und Methoden können hinzugefügt / verändert werden:**

```
const person = {
    name : "Michael",
    hallo : function(){
        return `Hallo ${this.name}`;
    }
};
person.hobby = "Hike";

person.hallo = function(){
    return `Hallo ${this.name} Hobby ${this.hobby}`;
};

console.log(person.hallo());
```

- **Auch von «Standard»-Objekten….**
  - Wichtig: Standardfunktionalität sollte nie verändert werden.

```
console.log("X");
console.log = value => {};
console.log("X");
```

FUNCTIONS

JavaScript Funktionen

- **Funktionen sind «First-Class Citizen»**
  - Können in Variablen abgespeichert werden
  - Können als Parameter übergeben werden
  - Können von Funktionen zurückgegeben werden
- **Besitzen eine offene Parameter-Liste**
  - Es können mehr oder weniger als die deklarierte Anzahl an Parameter übergeben werden.
  - Alle Parameter werden in *`arguments`* abgelegt.
- **Funktionen besitzen Properties**
- **Eine normale Funktion erzeugt einen eigenen Scope**

```
function helloWorld(a){
    console.log(a || "No Data");
}

function helloWorld2(){
    console.log(arguments[0]);
}

const sayHello = function(fnOutput)
{
    fnOutput("Hallo")
}

sayHello(helloWorld);
sayHello(helloWorld2);
```

Funktionen definieren

```
//Funktionen können definiert werden
function hallo(){
    console.log("Hallo");
}
hallo();

//Funktionen können einer Variable zugewiesen werden
const hallo2 = function(){
    console.log("Hallo2");
};
hallo2();

//Funktionen können einer Variable zugewiesen werden
const foo = hallo;
foo();
```

Funktionen als Parameter und Rückgabe

```
function add(a, b){
    return a + b;
}

function minus(a, b){
    return a - b;
}

function calc(fn, a ,b){
    console.log(fn(a,b));
}

calc(add, 3, 4);
calc(minus, 3, 4);
```

```
function addTo(a){
    return function(b){
        return a + b;
    }
}

const addTo3 = addTo(3);

addTo3(4);
```

Funktionen besitzen eine offene Parameter-Liste

- **arguments beinhaltet alle Parameter, welche der Funktion übergeben wurden**
- **arguments ist kein Array**
  - `Array.from(arguments);`

```
function foo(name){
    console.log(name);
    console.log(arguments.length);

    // Ausgabe aller Parameter...
    for(let i = 0; i< arguments.length; i++)
    {
        console.log(i, arguments[i]);
    }
    // ...oder als Array
    console.log(Array.from(arguments).join("\n"));
}

foo("Michael", "Gfeller", "HSR", "IFS");
```

Rest-Parameters

- **Mit «…name» kann man den letzten Parameter als «Rest-Parameter» definieren**
- **Dieser Parameter wird mit allen restlichen Werten abgefüllt**
- **Keine restlichen Parameter erzeugt ein leeres Array**

```
function foo(name, ...params){
    console.log(1,name);
    console.log(2,params.join(";"));
}

foo("Michael", "Gfeller", "HSR", "IFS");
```

Funktionen besitzen Properties

- **.name beinhaltet den Namen der Funktion**
  - Anonyme Methoden besitzen keinen «name»
  - Dieser Name wird für den Stacktrace genutzt
    - Moderne Browser loggen, falls kein Name angegeben wurde, den Variablenamen
- **.length beinhaltet die Anzahl Parameter der Funktion**

```
const fn1 = function(){ return "Michael" };
console.log(fn1.name);    //""


const fn2 = function name(){ return "Michael" };
console.log(fn2.name);    //"name"
console.log(fn2.length); //0


const fn3 = function name(name){ return name };
console.log(fn3.length); //1
```

Funktionen Overloading

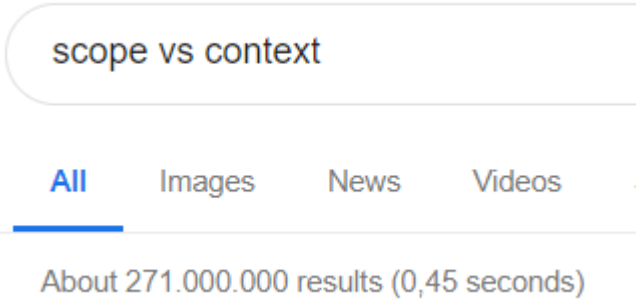
- **JavaScript kennt kein «Function Overloading»**
  - *Function Overloading: Gleicher Funktionsname mit unterschiedlichen Funktionsparameter*
- **Bei gleichen Funktionsnamen überschreibt die zuletzt definierte die vorhergehenden**
- **Lösung: Interne Weiche (typeof & arguments) oder sinnvolle default-Werte definieren**

```
function say(name) {
    console.log(`Hi ${name}!`);
}

function say() {
    console.log("Hi unknown Person!");
}

say("hi");
say("Michael");
```

```
function say(name = "unknown Person") {
    console.log(`Hi ${name}!`);
}

say();
say("Michael");
```

Funktionen Overloading – Beispiel

```
jQuery.fn.init = function( selector, context ) {
    //...
    if ( !selector ) {
        return this;
    }
    // Handle HTML strings
    if ( typeof selector === "string" ) {
        //...
    } else if ( selector.nodeType ) {
        //...
    } else if ( jQuery.isFunction( selector ) ) {
        //...
    }
    //...
    return jQuery.makeArray( selector, this );
};
```


scope vs context
All
Images
News
Videos
About 271.000.000 results (0,45 seconds)

SCOPE

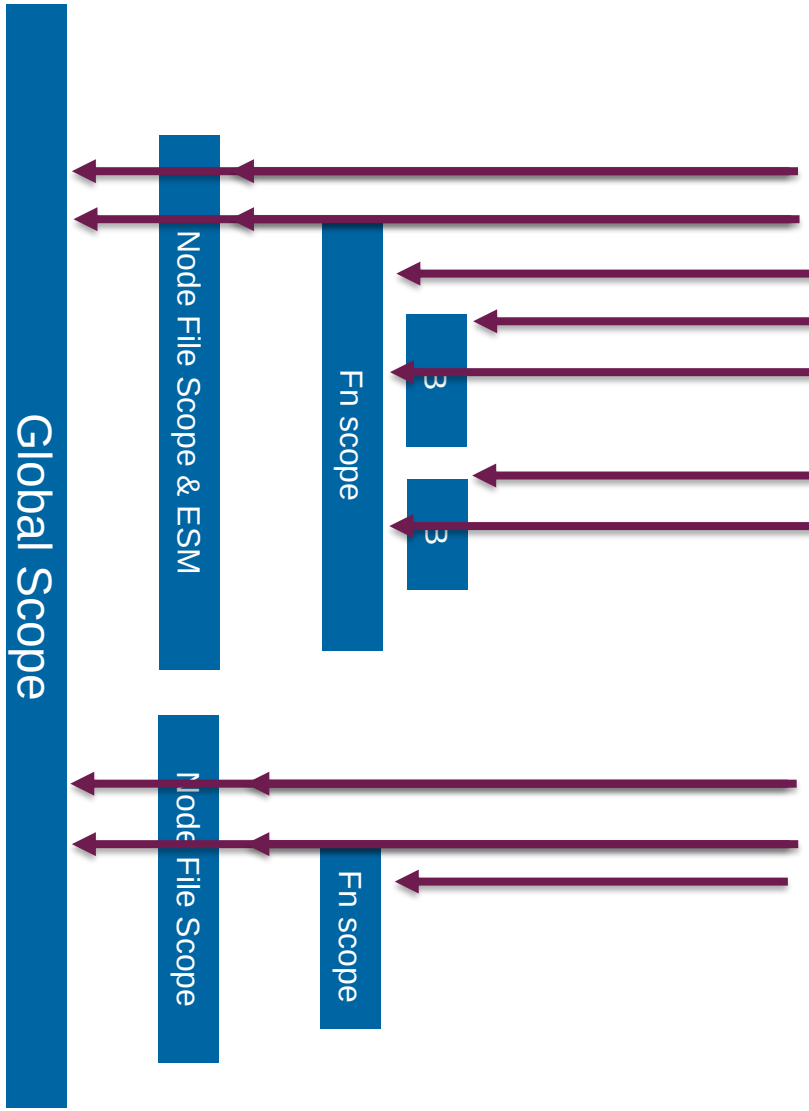
Scope

Scope – auf deutsch Geltungsbereich – bezeichnet allgemein einen Bereich, in dem etwas gültig oder verfügbar ist

Quelle: https://wiki.selfhtml.org/wiki/Scope

```
const funcA = function(){
    const a = 1;
    const funcB = function(){
        const b = 2;
        console.log(a, b); // outputs: 1, 2
    };
    funcB();
    console.log(a, b); // Error! b is not defined
};
funcA();
funcB(); // Error! funcB is not defined
```

Scope




```
//a.js
const a = "a";
function fnA() {
    const a1 = "a";
    for(let i=1; i < 10; i++){
        var r = i * 2;
    }
    for(let i=1; i < 10; i++){
        var r = i * 2;
    }
}

//b.js
const b = "b";
function fnb() {
    const b1 = "b"

}
```

B = Block Scope

Scope

- **Jede Funktion und jedes Objekt generiert einen neuen Scope**
- **Innerhalb von einem Scope kann man auf**
  - dessen Variablen
  - globale Variablen
  - Variablen aller «Parent»-Scopes zugreifen.
    - Dieses Feature nennt sich «Closure»
    - Diese Werte bleiben erhalten
- **Ein <script>-Tag erzeugt keinen Scope**
  - Auch bei externen Files gibt es keinen eigenen Scope
- **«ES modules» erzeugen einen Scope**
  - Nicht Teil dieser Vorlesung

Scope

- **Variable im globalen Scope**
  - Ohne «var» «let» «const»
  - Im Browser über window.myGlobalVariable zugreifbar. In node.js: global.myGlobalVariable
- **Variablen im lokalen / Funktionen Scope**
  - Benötigt ein «var» «let» «const»
  - Nur innerhalb des Scopes zugreifbar oder dessen «nested» Scopes
    - Closures
- **«Module Scope»**
  - Node.js & ES6-Module erzeugt pro File ein neuen Scope
    - Nicht **explizit** globale Variablen werden auf diesen gelegt. Dieser ist File basiert.
- **«Block Scopes»**
  - Mit «let» «const»

Scope

```
a = "A"; //wird auf das globale Objekt gelegt
var b = "B";  // wird auf den aktuellen Scope gelegt

function foo(){
    c = "C"; //wird auf das globale Objekt gelegt
    var d = "D"; // wird auf den aktuellen Scope gelegt
}
foo();

var globalObject = typeof(global) === "undefined" ? window : global; // browser oder node js
console.log(globalObject.a);
console.log(globalObject.b);
console.log(globalObject.c);
console.log(globalObject.d);
```

Node

```
A
undefined
C
undefined
```

Browser

Block Scope (ECMAScript 6)

- «let» und «const» als neues Keyword

```
'use strict'
for(let x = 1; x < 10; ++x){
    console.log(x);
    if(x > 4){
        let x = 10; //wird vom for-loop ignoriert
        // var x = 10 // Identifier 'x' has already been declared
    }
}
//console.log(x); //ReferenceError: x is not defined
```

Babeljs:

```
'use strict';
for (var _x = 1; _x < 10; ++_x) {
    console.log(_x);
    if (_x > 4) {
        var _x2 = 10; //wird vom for-loop ignoriert
    }
}
console.log(x); //ReferenceError: x is not defined
```

- Zuweisungen an «let» und «const» Definitionen werden nie aufs globale Objekt gelegt

```
var x = 2;
let y = 2;
console.log("x:", window.x);
console.log("y:",window.y);
```

```
x: 2
```

```
y: undefined
```

- Merke: In neuen Projekten «let» und «const» verwenden – «var» vermeiden.

CONTEXT

«this» Context

«this» ist der aktuelle Context.
«this» referenziert je nach Aufrufart ein anders Objekt:

- Falls ein eine Funktion als Methode von einem Objekt aufgerufen wird. Ist this = objekt
  Beispiel: **object.foo();**
- Falls eine Funktion mit new() aufgerufen wird. Wird «this» mit einem neu erstellten Objekt abgefüllt.
  Beispiel. **new foo();**
- Falls eine «unbound» Funktion aufgerufen wird. Zeigt «this» auf das globale Objekt.

Jeder Funktion kann mit apply() oder call() den Context gesetzt werden. In diesem fall werden die oben genannten Regeln ignoriert.

Z.B. **foo.call({ counter : 123});**

Jeder Funktion kann mit bind einen Context vorgeben werden. In diesem fall werden die oben genannten Regeln ignoriert.
Diese Regel wird ignoriert falls die «gebundene Funktion» mit new aufgerufen wird.

```
var boundFoo = foo.bind({counter : 11});
boundFoo();
```

Aufgabe: Scope

- **File: scope\scope1**
  - Überlegen Sie sich, was auf der Console ausgegeben wird. Überprüfen Sie Ihre Überlegung.
  - Würde dieses Script in einem Browser ausgeführt zum gleichen Resultat führen? Begründen Sie Ihre Antwort.
- **File: scope\scope2**
  - Ziel dieses Programms ist es, den Index der übergeben Buchstaben im Alphabet-Array zu finden und diese zurückzugeben.
  - Finden Sie die Ursache weshalb A funktioniert und B nicht.
    - Nutzen Sie den Debugger.
  - Ändern Sie den Code so, dass dieser wie vorgesehen funktioniert
- **Zeit: 20 Minuten**

Aufgabe: Context

- **File: context\context1**
  - Überlegen Sie sich, was auf der Console ausgegeben wird. Überprüfen Sie Ihre Überlegung.
- **File: functions\function1**
  - Überlegen Sie sich, was auf der Console ausgegeben wird. Überprüfen Sie Ihre Überlegung.
  - Führen Sie den Code aus und überprüfen Sie Ihre Überlungen.

- **Zeit: 15 Minuten**

USE STRICT

Strict Mode / ‘use strict’

‘use strict’ hat folgende Ziele

- **Eliminiert «fails silently»**
  - Falls eine Variable ohne «var» definiert wird
  - Falls Read-Only Werte gesetzt werden
    - Ohne Strict Mode: Wird einfach ignoriert
  - …
- **Eliminiert «Probleme» welchen es Compiler verunmöglicht den Code zu optimieren**
- **Security wird «leicht» verbessert**
  - Z.B. wird bei «unbound» Funktion «this» nicht auf das globale Objekt gelegt.

Wichtig: das Laufzeitverhalten ändert sich!

Details: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Strict_mode

Strict Mode aktivieren

- ‘use strict’ im JavaScript aktiviert den Strict Mode
  - Erste Linie einem File
    - Ganzes File wird in Strict Mode geschalten;
  - Erste Linie in einer Function
    - Funktion und nested Funktionen werden in Strict Mode geschaltet;
- Strict Mode wird vererbt
  - Alle «nested» Scopes werden auch Strict
  - Funktionen werden nicht in Strict Mode geschalten obwohl Sie von einer Strict Funktion aufgerufen werden
- Wichtig
  - Strict Mode kann nicht mehr entfernt werden
  - Strict Mode muss pro File aktiviert werden
  - «Strict Mode» JavaScript sollte nicht mit «Non Strict Mode» JavaScript zu einem JavaScript File vereint werden (concatenate)

```
function a() {
    a1 = 1; //ok
}

function c() {
    'use strict';

    function b() {
        b1 = 1; //error
    }
    b();
    a();
}
c();
```

Strict Mode Fazit

- Immer verwenden ausser:
  - Legacy Code
  - Die Files werden zusammengefügt
    - Vermischung von Legacy Code und Strict Code
    - Beim zusammenfügen kann meistens gewählt werden ob die erste Zeile mit ‘use strict’ ergänzt werden sollte.
      - Beispiel: https://www.npmjs.com/package/gulp-concat-util
        «Advanced usage example, replacing any 'use strict;' statement found in the files with a single one at the top of the file»

ARROW FUNCTION / ARROW

ECMAScript 6 Funktionen

- **ECMAScript 6 erlaubt es Funktionen mit einem «Arrow-Syntax» zu definieren**
  - Oft auch «Lambda» genannt (C# / Scala)
- **Sinnvoll für (sehr) kleine Funktionen z.B. als Filter Parameter**

```
[ 'a', 'b', 'c' ].forEach((elem, index) =>  console.log(index +":"+elem));

var array = [1,2,3,4].map(x => x*x);

var filteredArray = array.filter( elem => elem > 5);

console.log(array.every( elem => elem > 5));

console.log((()=>{
    var x = 9;
    var y = 11;
    return x + y;
})());
```

- **Bei Lambda-Funktionen ist der Context immer auf das selbe Objekt gebunden.**

```
function Point(x, y){
    this.x = x;
    this.y = y;
    this.area = () => this.x + this.y;
}

var areaFn = new Point(10,50).area;
console.log(areaFn());
```

babeljs

```
function Point(x, y) {
    var _this = this;

    this.x = x;
    this.y = y;
    this.area = function () {
        return _this.x + _this.y;
    };
}
```

JAVASCRIPT FEATURES

Hoisting

Variante 1:

```
"use strict";
var a
a = 1;
console.log("0",a);
```

Variante 2:

```
"use strict";
a = 1;
console.log("0",a);
var a;
```

- Dieses Feature nennt sich «Hoisting»
- JavaScript verschiebt alle Deklarationen von Methoden / Variablen an den Anfang des Scopes.
- Initialisierungen werden nicht ge-«hoisted»
- Kann zu Bugs führen.
- Funktionen-Definitionen werden auch verschoben.
- «let, const» werden ge-«hoisted» erzeugen aber eine «temporal dead zone».
- Achtung: Bei einer «Function-Expressions» wird die Zuweisung nicht verschoben.

Hoisting

	Hoisting	Scope	Creates global properties
var	Declaration	Function	Yes
let	Temporal dead zone	Block	No
const	Temporal dead zone	Block	No
function	Complete	Block	Yes
class	No	Block	No
import	Complete	Module-global	No

Quelle: http://exploringjs.com/es6/ch_variables.html#_ways-of-declaring-variables

Hoisting Aufgabe

Aufgabe:
Was geben die console.log()-Statements aus, mit Begründung:

```
var x = 1;
var y = 2;
function print(x) {
    console.log(x); //I
    console.log(y); //II
    var x = 3;
    var y = 4;
}
print();
```

```
let x = 1;
let y = 2;
function print(x) {
    console.log(x);  //III
    console.log(y);  //IV
    let x = 3;
    let y = 4;
}
print();
```

Zeit: 3 Minuten

Jedes Objekt ist eine HashTable

- **Jedes Objekt ist eine HashTable**
  - Ausnahme: null, undefined
- **Zugriff auf Eigenschaften vom Objekt mit**
  - obj[PropertyName] bzw. obj[PropertyName] = "A "
- **Objekte**
  - Wandeln den Index Wert immer in einen String.
    - Sample[1] == Sample["1"]
- **Array**
  - Unterstützt ganzzahlige Nummern als Indexer
  - Nicht ganzzahlige Nummern werden zu Strings gewandelt.
    - Das Array verhält sich wie ein normales Objekt
  - [].length beachtet nur die echten Array Einträge.
  - *Wichtig: Ein Array sollte wie ein Array verwendet werden.*
- **Functions**
  - Können wie Objekte mit Properties ergänzt werden

Statement oder Expression

Syntax

```
while (condition) {
  statement
}
```

condition

An expression evaluated before each pass through the loop. If this condition evaluates to true, `statement` is executed. When condition evaluates to false, execution continues with the statement after the `while` loop.

Statement oder Expression

- **Expression**
  - Erzeugt einen Wert
  - Kann als Parameter einer Funktion übergeben werden
  - Beispiele:
    - myfunc("a", "b")
    - 3 + x
    - myVar

```
let x = (y >= 0 ? y : -y);
```

- **Statement**
  - Führt etwas aus (Sprachelemente)
  - Beispiele:
    - If
    - Loops
    - var / let / const

```
let x;

if (y >= 0)
{
    x = y;
}
else {
    x = -y;
}
```

Statement oder Expression

```
function testExpression( a ) {
    console.log(a);
}
var i = 10
testExpression("A");
testExpression(a = i > 0 ? i : -i );
testExpression(var a = i > 0 ? i : -i );
testExpression((var a = i > 0 ? i : -i));
```

https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/var

Semicolon insertion

Details: https://tc39.github.io/ecma262/#sec-automatic-semicolon-insertion

- **A semicolon is inserted before, when a Line terminator or "}" is encountered that is not allowed by the grammar.**
  - Ein Semikolon wird eingefügt, falls die zusammengefügte Zeile zu einem Fehler führen würde.

```
let x = 10
let y = 20;
console.log(x + "," + y);
```

```
let x = 10 let y = 20;    // Grammar-Fehler
console.log(x + "," + y);
```

```
let a = 2 * 4
(a).toString();
```

```
let a = 2 * 4(a).toString();
```

Semicolon insertion

- **A semicolon is inserted at the end, when the end of the input stream of tokens is detected and the the parser is unable to parse the single input stream as a complete program.**
  - Semicolon am Ende eines Programmes
- **A semicolon is inserted at the end, when a statement with restricted productions in the grammar is followed by a line terminator.**
  - Statement wird mit dem nächsten “line terminator” beendet.
  - PostfixExpressions (++ and --); continue; break; return;

```
function createUser(name)
{
    return
    {
        name : name
    }
}
```

```
function createUser(name)
{
    return;
    {
        name : name
    }
}
```

Fazit: Semicolon insertion

- Falls Semikolons weggelassen werden, fügt der JavaScript-Interpreter diese ein.
- Automatisch gesetzte Semikolons können zu Bugs führen.
- Programmierer sollten sich NIE auf automatisch gesetzte Semikolons verlassen.
- Aber: Für bessere Übersicht sinnvoll:

```
var result = [4,2,1,5].map(function(x){ return x - 3}).filter(function(x){return x > 0}).sort();
console.log(result);
```

```
var result = [4,2,1,5]
    .map(function(x){ return x - 3})
    .filter(function(x){return x > 0})
    .sort();
```

SHIM / POLYFILL & SYMBOL

Shim / Polyfill

- **Unterschied zwischen Shim und Polyfill**
  - Confusing – viele erklärungen im Web!
  - Einfachste Erklärung: *A polyfill is a shim for a browser API* *(Quelle: https://en.wikipedia.org/wiki/Polyfill - Liste mit wichtigen Polyfills)*
- **Eigenschaften**
  - Implementieren API's die noch nicht zu Verfügung stehen.
  - Fixen JavaScript Bugs in alten Browsern
  - Polyfill können wieder entfernt werden – falls dieser nicht mehr notwendig ist.

Symbol

- **Details: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Symbol**
- **Beschreibung: *A symbol is a unique and immutable data type and may be used as an identifier for object properties.***
- **Aktueller-Einsatz**
  - Frameworks um “Hooks” zu definieren
  - Iterator (for of, spread)
- **Eigenschaften**
  - Exception beim Versuch den Typ zu Ändern
    - Ausnahme: String(Symbol(2))
- **Wichtig:**
  - Symbol ermöglichen es NICHT private Properties auf Objekten zu generieren
  - Es gibt kein JavaScript gegenstück in EcmaScript 5 => benötigt zusätzlichen Code; sogenannte Shims / Polyfills

UTILITIES

Regex

Ein regulärer Ausdruck (englisch regular expression, Abkürzung RegExp oder Regex) ist in der theoretischen Informatik eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient.
Quelle: Wikipedia

Teil der Sprachdefinition

- **Pattern-Matching**
- **Search & Replace**

Syntax: `/ muster / flags` ***bzw.*** `RegExp(muster, flags)`

Muster: Ausdruck, welcher gesucht werden sollte. Kann spezielle Zeichen beinhalten

Flags: g, i, m

Spezielle Zeichen: . \d \D \s …

Ausführlich: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Regex - Beispiele

Ersetze alle ‘JS‘ durch ‘JavaScript’

"Das ist ein Text über JS".replace(/JS/i , "JavaScript")

Erkennen ob der RegEx matched

/JS/i.test("Das ist ein Text über JS")

"Das ist ein Text über JS".match(/JS/)

Compile von Regex-Ausdrücken

```
var compiled = /JS/;
compiled.compile();
```

http://jsperf.com/regexcompile

Date

Date Objekt

- **Datums-Funktionalitäten**
- **Date() > erzeugt ein Datums-String**
- **new Date() => erzeugt ein Datums-Objekt**
- **new Date([YEAR], [MONTH], [DAY],…) => erzeugt ein Datums-Objekt mit den Daten**
  - new Date(2015, 20, 10) => 10 Sept. 2016

Date to String

- **Manuell mit getMonth() getDay() getYear()**
- **new Date().toLocaleDateString()**
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/toLocaleDateString
- **Package: http://momentjs.com/**

Math

Math Objekt

- **Mathe-Funktionalitäten**
- **Mathe-Konstanten wie PI**
- **Winkel-Funktionen in Radian**
- **Implementation ist Browser-spezifisch. D.h. unterschiedliche Genauigkeit je nach Browser.**

STYLEGUIDES

Eslint

Find and fix problems in your JavaScript code

- **Installation**
  - https://eslint.org/docs/user-guide/getting-started
  - Webstorm
    - https://www.jetbrains.com/help/webstorm/eslint.html
  - Visual Studio Code
    - https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint

Aufgabe JavaScript Code Guideline evaluieren

Nachfolgend sind ein paar öffentliche JS Style Guides aufgelistet:

- https://github.com/airbnb/javascript
  - Sehr zu empfehlen & wird gewartet.
- https://github.com/airbnb/javascript/tree/master/es5
- http://www.w3schools.com/js/js_conventions.asp
- https://google.github.io/styleguide/javascriptguide.xml
- https://www.w3.org/wiki/JavaScript_best_practices
- https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style

Aufgabe:

- **Analysieren Sie die Style Guides**
  - Überlegen Sie sich weshalb die Punkte erwähnt werden
  - Sind die Style Guides zu Umfangreich
  - Fehlen wichtige Punkte
- **Installieren Sie ESLint und aktivieren Sie es**
- **Zeit: 30 Minuten**

ZUM NACHLESEN

Zum Nachlesen

- https://exploringjs.com/impatient-js/toc.html
- **http://exploringjs.com/es6/**
- **http://speakingjs.com/es5/index.html**
- **http://exploringjs.com/es2016-es2017.html**
- **http://es6-features.org**